Attorney's Docket No.: 10559-613001 / P12852

APPLICATION

FOR

UNITED STATES LETTERS PATENT

TITLE: CONTROL MECHANISMS FOR ENQUEUE AND DEQUEUE OPERATIONS IN A PIPELINED NETWORK PROCESSOR

APPLICANT: GILBERT WOLRICH, MARK B. ROSENBLUTH, DEBRA BERNSTEIN, AND MATTHEW J. ADILETTA

CERTIFICATE OF MAILING BY EXPRESS MAIL

Express Mail Label No. EL870691199US

I hereby certify that this correspondence is being deposited with the United States Postal Service as Express Mail Post Office to Addressee with sufficient postage on the date indicated below and is addressed to the Commissioner for Patents, Washington, D.C. 20231.

12-18-01
Date of Deposit

Signature

Gabe Lewis
Typed or Printed Name of Person Signing Certificate

CONTROL MECHANISMS FOR ENQUEUE AND DEQUEUE OPERATIONS IN A PIPELINED NETWORK PROCESSOR

BACKGROUND

This invention relates to control mechanisms for enqueue and dequeue operations in a pipelined network processor.

A network processor should be able to store newly received packets to a memory structure at a rate at least as high as the arrival time of the packets. To avoid dropping packets and still maintain system throughput, a packet should be removed from memory and also transmitted at the packet arrival rate. Thus, in the time it takes for a packet to arrive, the processor must perform two operations: a store operation and a retrieve from memory operation. The ability to support a large number of queues in an efficient manner is essential for a network processor connected to a high line rate network.

System designs based on ring data structures use statically allocated memory addresses for packet buffering and may be limited in the number of queues that can be supported. Systems that use linked lists are more flexible and allow for a large number of queues. However, linked list queues typically involve locking access to a queue descriptor and queue pointers when a dequeue request is made while an enqueue operation is in progress. Similarly, access to a queue descriptor and queue pointers is typically locked when an enqueue request is made while a dequeue operation is in progress or when near simultaneous enqueue operations or near simultaneous dequeue operations are made to the same queues. Therefore, for network processors connected to high line rates when the network traffic is being directed at a small subset of the available queues, the

latency to enqueue or dequeue packets from the same queue may be too great using atomic memory operators.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 illustrates a system that includes a pipelined network processor.

FIG. 2 illustrates a pipelined network processor.

FIG. 3 is a block diagram of a cache data structure to illustrate enqueue and dequeue operations.

FIG. 4 illustrates the flow of enqueue requests to a queue.

FIG. 5 is a block diagram showing an enqueue operation.

FIG. 6 illustrates the flow of dequeue requests to a queue.

FIG. 7 is a block diagram showing a dequeue operation.

DETAILED DESCRIPTION

Referring to FIG. 1, a network system 10 for processing data packets includes a source of data packets 12 coupled to an input of a network device 14. An output of the network device 14 is coupled to a destination of data packets 16. The network device 14 can include a network processor 18 with memory data structures configured to store and forward the data packets efficiently to a specified destination. Network device 14 can include a network switch, a network router or other network device. The source of data packets 12 can include other network devices connected over a communications path operating at high data packet transfer line speeds such as an optical carrier line (e.g., OC-192), 10 Gigabit line, or other line speeds. The destination of data packets 16 can include a similar network connection.

Referring to FIG. 2, the network processor 18 has multiple programming engines that function as a receive pipeline 21, a

transmit scheduler 24, a queue manager 27 and a transmit pipeline 28. Each programming engine contains a multiple-entry content addressable memory (CAM) to track N of the most recently used queue descriptors where N represents the number of entries contained in the CAM. For example, the queue manager 27 includes the CAM 29. The network processor 18 includes a memory controller 34 that is coupled to a first 30 and second memory 32, and a third memory 17 containing software instructions for causing the engines to operate as discussed in detail below. The memory controller 34 initiates queue commands in the order in which they are received and exchanges data with the queue manager 27. The first memory 30 has a memory space for storing data. The second memory 32 can be coupled to the queue manager 27 and other components of the network processor 18. As shown in FIG. 2, the first memory 30 and the second memory 32 reside externally to the network processor 18. Alternatively, the first memory 30 and/or the second memory 32 can be internal to the network processor 18. The processor 18 also includes hardware interfaces to a receive bus and a transmit bus that are coupled to receive and transmit buffers 20, 36.

A receive buffer 20 is configured to buffer data packets received from the source of data packets 12. Each data packet can contain a real data portion representing the actual data being sent to the destination, a source data portion representing the network address of the source of the data, and a destination data portion representing the network address of the destination of the data. The receive pipeline 21 is coupled to the output of the receive buffer 20. The receive pipeline 21 also is coupled to a receive ring 22, which may have a first-in-first-out (FIFO) data structure. The receive ring 22 is coupled to the queue manager 27.

The receive pipeline 21 makes enqueue requests 23 to the queue manager 27 through the receive ring 22. The receive pipeline 21 can include multiple multi-threaded programming engines working in a pipelined manner. The engines receive packets, classify them, and store them on an output queue based on the classification. The receive processing determines an output queue for each packet. By pipelining, the programming engine can perform the first stage of execution of an instruction and when the instruction passes to the next stage, a new instruction can be started. The processor does not have to lie idle while waiting for all steps of the first instruction to be completed. Therefore, pipelining can lead to improvements in system performance.

The receive pipeline 21 can be configured to process the data packets from the receive buffer 20 and store the data packets in a data buffer 38 in the memory 32. Once the data packets are processed, the receive pipeline 21 generates enqueue requests 23 directed to the queue manager 27. Each enqueue request represents a request to append a newly received buffer to the last buffer in a queue of buffers 48 in the first memory 30. The receive pipeline 21 can buffer several packets before generating the enqueue requests. Consequently, the total number of enqueue requests generated can be reduced.

The transmit scheduler 24 is coupled to the queue manager 27 and is responsible for generating dequeue requests 25 based on specified criteria. Such criteria can include the time when the number of buffers in a particular queue of buffers reaches a predetermined level. The transmit scheduler 24 determines the order of packets to be transmitted. Each dequeue request 25 represents a request to remove the first buffer from a queue 48 (discussed in greater detail below). The transmit scheduler 24 also may include scheduling algorithms for generating dequeue

requests 25 such as "round robin", priority based or other scheduling algorithms. The transmit scheduler 24 may be configured to use congestion avoidance techniques such as random early detection (RED), which involves calculating statistics for the packet traffic. The transmit scheduler maintains a bit for each queue signifying whether the queue is empty or not.

The queue manager 27, which can include, for example, a single multi-threaded programming engine, processes enqueue requests from the receive pipeline 21 as well as dequeue requests from the transmit scheduler 24. The enqueue requests made by the receive pipeline and the dequeue requests made by the transmit scheduler may be present on the receive ring 22 before they are processed by the queue manager 27. The queue manager 27 allows for dynamic memory allocation by maintaining linked list data structures for each queue.

The queue manager 27 contains software components configured to manage a cache of data structures that describe the queues ("queue descriptors"). The cache has a tag portion 44a and a data store portion 44b. The tag portion 44a of the cache resides in the queue manager 27, and the data store portion 44b of the cache resides in a memory controller 34. The tag portion 44a is managed by the CAM 29 which can include hardware components configured to implement a cache entry replacement policy such as a least recently used (LRU) policy. The tag portion of each entry in the cache references one of the last N queue descriptors used to enqueue and dequeue packets by storing as a CAM entry that queue descriptor's location in memory, where N is the number of entries in the CAM. The corresponding queue descriptor is stored in the data store portion 44b of the memory controller 34 at the address entered in the CAM. The actual data placed on the queue is stored in the second memory 32.

The queue manager 27 can alternately service enqueue and dequeue requests. Each enqueue request references a tail pointer of an entry in the data store portion 44b. Each dequeue request references a head pointer of an entry in the data store portion 44b. Because the cache contains valid updated queue descriptors, the need to lock access to a queue descriptor 48a can be eliminated when near simultaneous enqueue and dequeue operations to the same queue are required. Therefore, the atomic accesses and latency that accompany locking can be avoided.

The data store portion 44b maintains a certain number of the most recently used (MRU) queue descriptors 46. Each queue descriptor includes pointers 49 to a corresponding MRU queue of buffers 48. In one implementation, the number of MRU queue descriptors 46 in the data store portion 44b is sixteen. Each MRU queue descriptor 46 is referenced by a set of pointers 45 residing in the tag portion 44a. In addition, each MRU queue descriptor 46 can be associated with a unique identifier so that it can be identified easily. Each MRU queue 48 has pointers 53 to the data buffers 38 residing in the second memory 32. Each data buffer 38 may contain multiple data packets that have been processed by the receive buffer 20.

The uncached queue descriptors 50 reside in the first memory 30 and are not currently referenced by the data store portion 44b. Each uncached queue descriptor 50 also is associated with a unique identifier. In addition, each uncached queue descriptor 50 includes pointers 51 to a corresponding uncached queue of buffers 52. In turn, each uncached queue 52 contains pointers 57 to data buffers 38 residing in the second memory 32.

Each enqueue request can include an address pointing to the data buffer 38 associated with the corresponding data

packets. In addition, each enqueue or dequeue request includes an identifier specifying either an uncached queue descriptor 50 or a MRU queue descriptor 46 associated with the data buffer 38.

In response to receiving an enqueue request, the queue manager 27 generates an enqueue command 13 directed to the memory controller 34. The enqueue command 13 may include information specifying a MRU queue descriptor 46 residing in the data store portion 44b. In that case using the pointer 49, the queue 48 is updated to point to the data buffer 38 containing the received data packet. In addition, the MRU queue descriptor 46 is updated to reflect the state of the MRU queue 48. The MRU queue descriptor 46 can be updated quickly and efficiently because the queue descriptor is already in the data store portion 44b.

If the enqueue command 13 includes a queue identifier specifying a queue descriptor which is not a MRU queue descriptor 46, the queue manager 27 replaces a particular MRU queue descriptor 46 with the uncached queue descriptor 50. As a result, the uncached queue descriptor 50 and the corresponding uncached queue of buffers 52 are referenced by the data store portion 44b. In addition, the newly referenced uncached queue 52 associated with the uncached queue descriptor 50 is updated to point to the data buffer 38 storing the received data packet.

In response to receiving a dequeue request 25, the queue manager 27 generates a dequeue command 15 directed to the memory controller 34. As with the enqueue commands 13 discussed above, each dequeue command 15 includes information specifying a queue descriptor. If a MRU queue descriptor 46 is specified, then data buffers 38 pointed to by a corresponding pointer 53 are returned to the queue manager 27 for further processing. The queue 48 is updated and no longer points to the returned data

buffer 38 because it is no longer referenced by the data store portion 44b.

The dequeue command 15 may include a queue descriptor which is not a MRU queue descriptor. In that case, the queue manager 27 replaces a particular MRU queue descriptor with the uncached queue descriptor. The replaced queue descriptor is written back to the first memory 30. As a result, the replacement MRU queue descriptor 46 and the corresponding MRU queue 48 are referenced by the data store portion 44b. The data buffer 38 pointed to by the queue 48 is returned to the queue manager 27 for further processing. The MRU queue buffer 48 is updated and no longer points to the data buffer 38 because it is no longer referenced by the data store portion 44b.

Referring to FIG. 3, the operation of the cache is illustrated. In this example, the tag portion 44a can contain sixteen entries. For purposes of illustration only, the following discussion focuses on the first entry in the tag portion 44a. The first entry is associated with a pointer 45a that points to a MRU queue descriptor 46a residing in the data store portion 44b. The queue descriptor 46a is associated with a MRU queue 48a. The queue descriptor 46a includes a head pointer 49a pointing to the first buffer A and a tail pointer 49b pointing to the last buffer C. An optional count field 49c maintains the number of buffers in the queue of buffers 48a. In this case the count field 49c is set to the value "3" representing the buffers A, B and C. As discussed in further detail below, the head pointer 49a, the tail pointer 49b and the count field 49c may be modified in response to enqueue requests and dequeue requests.

Each buffer in the queue 48a, such as a first buffer A, includes a pointer 53a to a data buffer 38a in the second memory 32. Additionally, a buffer pointer 55a points to a next ordered

buffer B. The buffer pointer 55c associated with the last buffer C has a value set to NULL to indicate that it is the last buffer in the queue 48a.

As shown in FIGS. 4 and 5, in response to the receiving an enqueue request 23, the queue manager 27 generates 100 an enqueue command 13 directed to the memory controller 34. In the illustrated example, the enqueue request 23 is associated with a subsequent data buffer 38d received after data buffer 38c. The enqueue request 23 includes information specifying the queue descriptor 46a and an address associated with the data buffer 38d residing in the second memory 32. The tail pointer 49b currently pointing to buffer C in the queue 48a is returned to the queue manager 27. The enqueue request 23 is evaluated to determine whether the queue descriptor associated with the enqueue request is currently in the data store portion 44b. If it is not, then a replacement function is performed 110. The replacement function is discussed further below.

The buffer pointer 55c associated with buffer C currently contains a NULL value indicating that it is the last buffer in the queue 48a. The buffer pointer 55c is set 102 to point to the subsequent buffer D. That is accomplished by setting the buffer pointer 55c to the address of the buffer D.

Once the buffer pointer 55c has been set, the tail pointer 49b is set 104 to point to buffer D as indicated by dashed line 61. This also may be accomplished by setting the tail pointer to the address of the buffer D. Since buffer D is now the last buffer in the queue 48a, the value of the buffer pointer 55d is set to the NULL value. Moreover, the value in the count field 49c is updated to "4" to reflect the number of buffers in the queue 48a. As a result, the buffer D is added to the queue 48a by using the queue descriptor 46a residing in the data store portion 44b.

The processor 18 can receive 106 a subsequent enqueue request associated with the same queue descriptor 46a and queue 48a. For example, it is assumed that the queue manager 27 receives a subsequent enqueue request associated with a newly arrived data buffer 38e. It also is assumed that the data buffer 38e is associated with the queue descriptor 46a. The tail pointer 49b can be set 108 to point to buffer E. That is represented by the dashed line 62 pointing to buffer E. The tail pointer 49b is updated without having to retrieve it because it is already in the data store portion 44b. As a result, the latency of back-to-back enqueue operations to the same queue of buffers can be reduced. Hence, the queue manager can manage requests to a large number of queues as well as successive requests to only a few queues or to a single queue. Additionally, the queue manager 27 issues commands indicating to the memory controller 34 which of the multiple data store portion entries to use to perform the command.

In some situations, however, none of the queue descriptors 46a currently occupying the data store portion 44b is associated with the newly arrived data buffer 38e. In that case, the processor performs 110 a replacement function removes a particular queue descriptor from the data store portion 44b according to a replacement policy. The replacement policy can include, for example, using a LRU policy in which a queue descriptor that has not been accessed during a predetermined time period is removed from the data store portion 44b. The removed queue descriptor is written back to the first memory 30. As discussed above, the removed queue descriptor is replaced with the queue descriptor associated with data buffer 38e. Once the replacement function is completed, queue operations associated with the enqueue request are performed as previously discussed above.

As shown in FIGS. 6 and 7, in response to receiving 200 a dequeue request, the queue manager 27 generates 200 a dequeue command directed to the memory controller 34. In this example, the dequeue request is associated with the queue descriptor 46a and represents a request to retrieve the data buffer 38a from the second memory 32. Once the data buffer 38a is retrieved, it can be transmitted from the second memory 32 to the transmit buffer 36. The dequeue request 25 includes information specifying the queue descriptor 46a. The head pointer 49a of the queue descriptor 46a points to the first buffer A which in turn points to data buffer 38a. As a result, the data buffer 38a is returned to the queue manager 27.

The head pointer 49a is set 202 to point to the next buffer B in the queue 48a as indicated by the dashed line 64. That can be accomplished by setting the head pointer 49a to the address of buffer B. The value in the count field 49c is updated to "4", reflecting the remaining number of buffers (B through E). As a result, the data buffer 38a is retrieved from the queue 48a by using the queue descriptor 46a residing in the data store portion 44b.

The queue manager 27 can receive 204 subsequent dequeue requests 25 associated with the same queue descriptor 46a. It is assumed, for example, that the queue manager 27 receives a further dequeue request 25 associated with the queue descriptor 46a. As indicated by the dashed line 64, the head pointer 46a currently points to buffer B which is now the first buffer because the reference to buffer A was removed. It also is assumed that the data buffer B is associated with queue descriptor 46a. The head pointer 49a can be set 206 to point to buffer C, as indicated by a dashed line 65, without having to retrieve the head pointer 49a because it is already in the data

store portion 44b. As a result, the latency of back-to-back dequeue operations to the same queue of buffers can be reduced.

In some situations, however, the queue descriptor 46a currently occupying an entry of the data store portion 44b is not associated with the data buffer 38b. In that case, the processor performs 208 a replacement function similar to the one discussed above. Once the replacement function has been completed, operations associated with the dequeue request are performed as previously discussed above.

The cache of queue descriptors can be implemented in a distributed manner such that the tag portion 44a resides in the memory controller 34 and the data store portion 44b resides in the first memory 30. Data buffers 38 that are received from the receive buffer 20 can be processed quickly. For example, the second of a pair of dequeue commands can be started once the head pointer for that queue descriptor is updated as a result of the first dequeue memory read of the head pointer. Similarly, the second of a pair of enqueue commands can be started once the tail pointer for that queue descriptor is updated as a result of the first enqueue memory read of the tail pointer. In addition, using a queue of buffers, such as a linked list of buffers, allows for a flexible approach to processing a large number of queues. Data buffers can be quickly enqueued to the queue of buffers and dequeued from the queue of buffers.

Various features of the system can be implemented in hardware, software, or a combination of hardware and software. For example, some aspects of the system can be implemented in computer programs executing on programmable computers. Each program can be implemented in a high level procedural or object-oriented programming language to communicate with a computer system. Furthermore, each such computer program can be stored on a storage medium, such as read-only-memory (ROM) readable by

a general or special purpose programmable computer, for configuring and operating the computer when the storage medium is read by the computer to perform the functions described above.

Other embodiments are within the scope of the following claims.